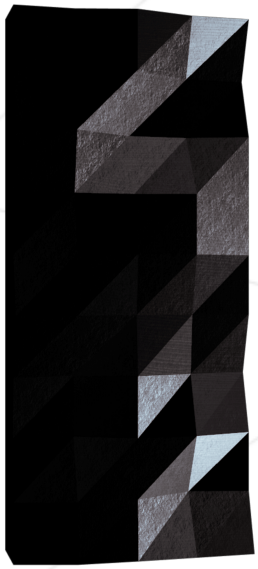


ft_containers

C++ containers, easy mode

Summary:
The standard C++ containers have all a specific usage.
To make sure you understand them, let's re-implement them!

Version: 4

Contents

Chapter I

Objectives

In this project, you will implement a few container types of the C++ standard template library.

You have to take the structure of each standard container as reference. If a part of the Orthodox Canonical form is missing in it, do not implement it.

As a reminder, you have to comply with the C++98 standard, so any later feature of the containers **MUST NOT** be implemented, but every C++98 feature (even deprecated ones) is expected.

Chapter II

General rules

Compiling

- Compile your code with `c++` and the flags `-Wall -Wextra -Werror`
- Your code should still compile if you add the flag `-std=c++98`

Formatting and naming conventions

- For each container, turn in the appropriately named class files.
- *Goodbye Norminette!* No coding style is enforced. You can follow your favorite one. But keep in mind that a code your peer-evaluators can't understand is a code they can't grade. Do your best to write a clean and readable code.

Allowed/Forbidden

You are not coding in C anymore. Time to C++! Therefore:

- You are allowed to use everything from the standard library. Thus, instead of sticking to what you already know, it would be smart to use as much as possible the C++-ish versions of the C functions you are used to.
- However, you can't use any other external library. It means C++11 (and derived forms) and `Boost` libraries are forbidden. The following functions are forbidden too: `*printf()`, `*alloc()` and `free()`. If you use them, your grade will be 0 and that's it.

A few design requirements

- Memory leakage occurs in C++ too. When you allocate memory, you must avoid **memory leaks**.
- Any function implementation put in a header file (except for function templates) means 0 to the exercise.
- You should be able to use each of your headers independently from others. Thus, they must include all the dependencies they need. However, you must avoid the problem of double inclusion by adding **include guards**. Otherwise, your grade will be 0.

Read me

- You can add some additional files if you need to (i.e., to split your code) and organize your work as you wish as long as you turn in the mandatory files.
- By Odin, by Thor! Use your brain!!!

Since your task here is to recode the STL containers, you of course **cannot use them** in order to implement yours.

Chapter III

Mandatory part

Implement the following containers and turn in the necessary `<container>.hpp` files:

- `vector`
  You don't have to do the `vector<bool>` specialization.
- `map`
- `stack`
  It will use your `vector` class as default underlying container. But it must still be compatible with other containers, the STL ones included.

You can pass this assignment without the stack (80/100).
But if you want to do the bonus part, you have to implement the 3 mandatory containers: vector, map and stack.

You also have to implement:

- `iterators_traits`
- `reverse_iterator`
- `enable_if`
  Yes, it is C++11 but you will be able to implement it in a C++98 manner.
  This is asked so you can discover SFINAE.
- `is_integral`
- `equal` and/or `lexicographical_compare`
- `std::pair`
- `std::make_pair`

III.1 Requirements

- The namespace must be `ft`.
- Each inner data structure used in your containers must be logical and justified (this means using a simple array for `map` is not ok).
- You cannot implement more public functions than the ones offered in the standard containers. Everything else must be private or protected. Each public function or variable must be **justified**.
- All the member functions, non-member functions and overloads of the standard containers are expected.
- You must follow the original naming. Take care of details.
- If the container has an **iterator** system, you must implement it.
- You must use `std::allocator`.
- For non-member overloads, the keyword `friend` is allowed. Each use of `friend` must be justified and will be checked during evaluation.
- Of course, for the implementation of `map::value_compare`, the keyword `friend` is allowed.

You can use https://www.cplusplus.com/
and https://cppreference.com/ as references.

III.2 Testing

- You must also provide tests, at least a `main.cpp`, for your defense. You have to go further than the `main` given as example!
- You must produce two binaries that run the same tests: one with your containers only, and the other one with the STL containers.
- Compare **outputs** and **performance / timing** (your containers can be up to 20 times slower).
- Test your containers with: `ft::<container>`.

A main.cpp file is available to download on the intranet project page.

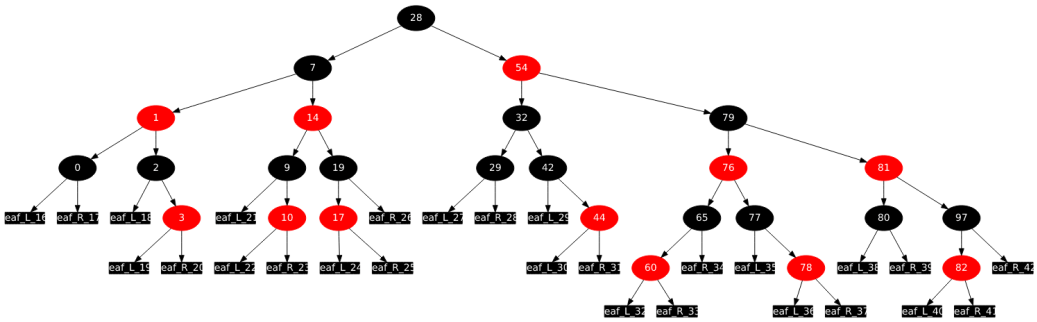
Chapter IV

Bonus part

You will get extra points if you implement one last container:

- `set`

But this time, a **Red-Black tree** is **mandatory**.



The bonus part will only be assessed if the mandatory part is PERFECT. Perfect means the mandatory part has been integrally done and works without malfunctioning. If you have not passed ALL the mandatory requirements, your bonus part will not be evaluated at all.

Chapter V

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your files to ensure they are correct.